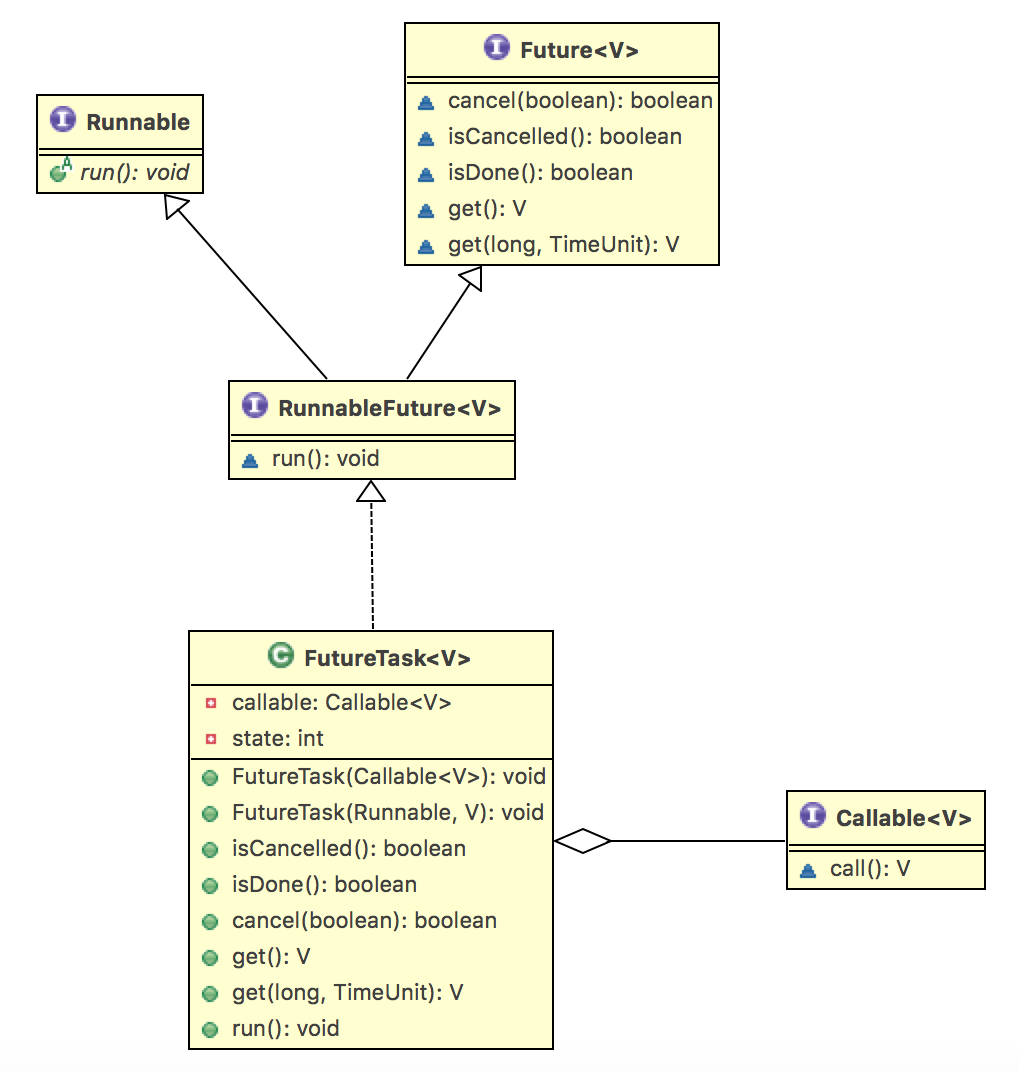
FutureTask实现原理

本文将剖析带有返回值的线程实现方式。FutureTask类关系如下：



首先看FutureTask的两个构造方法：

//构造方法一

public FutureTask(Callable<V> callable) {

if (callable == null)

throw new NullPointerException();

this.callable = callable;

this.state = NEW; // ensure visibility of callable

}

//构造方法二

public FutureTask(Runnable runnable, V result) {

//将runnable包装成callable

this.callable = Executors.callable(runnable, result);

this.state = NEW; // ensure visibility of callable

}

第一个构造方法我们比较熟悉，第二个构造方法可以用Runnable构造FutureTask，将Runnable使用适配器模式构造成FutureTask，使其具有FutureTask的特性，如可在主线程捕获Runnable的子线程异常。

构造完FutureTask，就可以用FutureTask构造Thread，并启动线程。启动线程会调用FutureTask的run()方法，run()方法是FutureTask的实现关键：

public void run() {

if (state != NEW || !UNSAFE.compareAndSwapObject(this, runnerOffset, null, Thread.currentThread()))

return;

try {

Callable<V> c = callable;

if (c != null && state == NEW) {

V result;

boolean ran;

try {

result = c.call();//1、获取返回值

ran = true;

} catch (Throwable ex) {

result = null;

ran = false;

setException(ex); //2、FutureTask的异常处理关键

}

if (ran)

set(result); //3、设置返回值

}

} finally {

// runner must be non-null until state is settled to prevent concurrent calls to run()

runner = null;

// state must be re-read after nulling runner to prevent leaked interrupts

int s = state;

if (s >= INTERRUPTING)

handlePossibleCancellationInterrupt(s);

}

}

protected void setException(Throwable t) {//异常处理

if (UNSAFE.compareAndSwapInt(this, stateOffset, NEW, COMPLETING)) {

outcome = t;

UNSAFE.putOrderedInt(this, stateOffset, EXCEPTIONAL); //设置为异常状态

finishCompletion();

}

}

protected void set(V v) {//设置正常返回值

if (UNSAFE.compareAndSwapInt(this, stateOffset, NEW, COMPLETING)) {

outcome = v;

//设置为正常结束状态

UNSAFE.putOrderedInt(this, stateOffset, NORMAL); // final state

finishCompletion();

}

}

在run()方法中，会调用callable对象的call()方法，并获取方法返回值，同时对call()方法中的异常进行了处理。异常时会将outcome设置为抛出的异常，正常时会将outcome设置为正常返回值，并将state设置成相应的状态。

run()分析完，下一步就要分析future.get()获取线程返回结果时如何工作。

public V get() throws InterruptedException, ExecutionException {

int s = state;

//未完成，则进入阻塞状态，等待完成

if (s <= COMPLETING)

s = awaitDone(false, 0L);

return report(s); //判断处理返回值

}

private V report(int s) throws ExecutionException {

Object x = outcome;

//根据state判断线程处理状态，并对outcome返回结果进行强转。

if (s == NORMAL)

return (V)x;

if (s >= CANCELLED)

throw new CancellationException();

throw new ExecutionException((Throwable)x); //在主线程中抛出异常

}

分析完run()方法和get()方法，其实对于FutureTask的返回值获取原理有了基本了解。下面继续分析其他要点：

线程状态

//FutureTask定义的7种线程状态

private volatile int state;

private static final int NEW = 0;

private static final int COMPLETING = 1; //设置返回值的过程，这个状态很短，可以划分为已完成状态。参考isDone()方法；

private static final int NORMAL = 2;

private static final int EXCEPTIONAL = 3;

private static final int CANCELLED = 4;

private static final int INTERRUPTING = 5;

private static final int INTERRUPTED = 6;

//是否已取消

public boolean isCancelled() {

return state >= CANCELLED;

}

//是否已完成

public boolean isDone() {

return state != NEW;

}

线程的状态在执行过程不同阶段不断变化，这是FutureTask的状态控制关键。注意state是volatile修饰，保障了多线程间的可见性。

阻塞等待

线程status未NEW和COMPLETING的时候，会进入awaitDone方法，表示要等待完成。awaitDone方法如下：

private int awaitDone(boolean timed, long nanos)throws InterruptedException {

final long deadline = timed ? System.nanoTime() + nanos : 0L;

WaitNode q = null;

boolean queued = false;

for (;;) {

if (Thread.interrupted()) {//线程是否被打断

removeWaiter(q);

throw new InterruptedException();

}

int s = state;

if (s > COMPLETING) {//已完成

if (q != null)

q.thread = null;

return s;

} //正在处理返回值，这里时间很短，所以调用Thread.yield()方法，短时间的线程让步。

else if (s == COMPLETING) // cannot time out yet

Thread.yield();

else if (q == null) //创建等待节点

q = new WaitNode();

else if (!queued) //CAS把该线程加入等待队列

queued = UNSAFE.compareAndSwapObject(this, waitersOffset, q.next = waiters, q);

else if (timed) { //超时等待

nanos = deadline - System.nanoTime();

if (nanos <= 0L) {

removeWaiter(q);

return state;

}

LockSupport.parkNanos(this, nanos); //阻塞一段时间

}

else

LockSupport.park(this); //线程阻塞，等待被唤醒

}

}

整个awaitDone的流程，暗含很多优化逻辑，值得思考。

唤醒

private void finishCompletion() {

// assert state > COMPLETING;

for (WaitNode q; (q = waiters) != null;) {

if (UNSAFE.compareAndSwapObject(this, waitersOffset, q, null)) {

for (;;) {

Thread t = q.thread;

if (t != null) {

q.thread = null;

LockSupport.unpark(t); //唤醒线程

}

WaitNode next = q.next;

if (next == null)

break;

q.next = null; // unlink to help gc

q = next;

}

break;

}

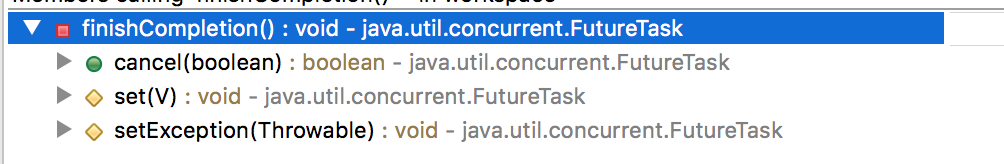
}

done();

callable = null; // to reduce footprint

}

finishCompletion()会在以下三处被调用：



在任务被取消、正常完成或执行异常时会调用finishCompletion()方法，从而唤醒等待队列中的线程。